UPPSALA
UNIVERSITET

IT 12 059
Examensarbete 15 hp
November 2012

Performance Comparisons of IP Problem Formulation

Joakim Lindqvist

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Performance Comparisons of IP Problem Formulation

Joakim Lindqvist

When solving optimization problems, the importance of speed can not be emphasized enough for many organizations. One company encountered a major performance difference when solving a problem with the same integer programming solver, in two different locations. The difference was shown not to be caused by the environment of the solver, but rather a reformulation of the problem. However, the reformulation did not improve the performance of an expanded version of the problem. By analyzing and comparing the two versions one might be able to find the properties of a problem which enables the reformulation to reduce the solving time. This in turn can be used to identify for which problems the reformulation should be applied to increase the speed at which they are solved.

Handledare: Arne Andersson
Ämnesgranskare: Pierre Flener
Examinator: Olle Gällmo
IT 12 059
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Today it is standard procedure to use computers to aid in the decision making process and for many companies the speed of acquiring an answer can not be emphasized enough. This can result in complex problems that do not only require fast computers but also advanced algorithms to be solved fast. There are several programs, so called solvers, that fill this need.

But, for most solvers, the solving time can be affected by the way that the problem is presented to it, i.e. how it is formulated. This report describes a case where a reformulation of a problem resulted in a drastic reduction in solving time.

The text is written for someone who has basic knowledge in maths and computer science at Bachelor level. This report will go through Linear and Integer programming for those unfamiliar with the subject as well as provide some descriptions of ways to reformulate Integer programming problems.

The report then continues with the aforementioned case. The section starts with a short brief of Xpress solver, the integer programming solver used, and then goes on describing the intricacies of the case and specific steps taken to analyse it.

For example, a parser was created to handle special problem formats (LP-files) and a description of the it can be found in section 4 which can be used as a reference when studying the code or to get a brief understanding of a LP file and its different formats.

The findings of the analysis is presented last in the document together with some conclusions drawn from them. A short discussion is then given which explains how the results of this project can be used in improving the solving time and what further work in the area might be.

2 Background

This section provides a foundation for reading the rest of the report. It begins with a description of linear programming (LP); its history, definition and uses which is followed by one of integer programming (IP) as well. Then the branch and bound method for solving integer programming problems is explained and the chapter ends with some ways to formulate these kinds of problems.

2.1 Linear Programming

The beginning of this section is about the history of linear programming. Then comes a description about the LP problem followed by an example of it.

2.1.1 History

Linear programming is a special case of mathematical programming. Mathematical programming (or optimization) problems are concerned, as defined by Gass[1] [1], with *the efficient use or allocation of limited resources to meet desired objectives.*

LP finds an optimal solution for a given mathematical model of a problem which is defined by linear relationships. One of the first ways of solving LP problems, the Simplex method, was published by Dantzig and its importance can be summed up by the writing of J. C. Nash [2]

> *In 1947, George Dantzig created a simplex algorithm to solve linear programs for planning and decision-making in large-scale enterprises. The algorithm's success led to a vast array of specialization and generalization that have dominated practical operations research for half a century.*

Although important, the Simplex method's worst case complexity was proven to be in exponential time in 1979 by V. Klee and G. J. Minty [3]. Despite this, the Simplex method is, in a revised form, still used today [4] and was listed in the top 10 algorithms by the journal *Computing in Science and Engineering* [2].

Many problems can be expressed with this kind of model which has led to LP being used in a wide range of areas. From the scheduling of petroleum refineries [6], which actually was one of the first industrial applications of LP [7], to the optimal routing of messages in a communication network [8].

V. Chvátal [9] notes that in 1975 the Nobel Prize in economical science was awarded to L. V. Kantorovich and T. C. Koopmans "for their contribution to the theory of optimum allocation of resources". He further remarks that the reason for it not going to Dantzig (by some considered the father of LP) might be that his work was too mathematical (and there is no Nobel Prize in mathematics).

[1] Although Gass himself called it a "programming problem", which comes from the use of the United States military, at that time, to refer to proposed training or logistics schedules [5].

2.1.2 The LP Problem

The LP problem can, more formally, be defined as by Ferguson [10] as the problem of *maximizing or minimizing a linear function subject to linear constraints*. Where the constraints may be equalities or inequalities. This can be written as:

maximize or minimize

$$z = \sum_{j=i}^{p} c_j x_j \tag{1}$$

subject to

$$\sum_{j=i}^{p} a_{ij} x_j \ (\leq, = \text{ or } \geq)\ b_i, \tag{2}$$

$$x_j \geq 0,$$

for $i = \{1, 2, \ldots, m\}$ and $j = \{1, 2, \ldots, p\}$. Where c_j, a_{ij} and b_i are known constants for all i and j, and x_j are nonnegative variables. (1) is called the objective function and (2) are called the constraints.

H. A. Taha shows in his book [11, p.35] that this problem can be made into equations by adding or subtracting nonnegative variables, so called *slack variables*. If the ith inequality is $\leq$ then add x_{p+i} and if the kth inequality is $\geq$ then subtract x_{p+k}. The problem can then be put into matrix-form as:

maximize or minimize

$$z = CX$$

subject to

$$AX = B,$$
$$X \geq 0,$$

where C is a n-row vector, A is an $m \times n$ matrix and X and B are two m-column vectors.

A LP problem either has an optimal solution or one does not exist. If it exists then it is either an *unique optimal solution* or there are *alternative solutions* (infinitely many - due to linearity). Although after obtaining an optimal solution one rarely tries to find the alternative ones.

The problem can also be *infeasible*. That is, all the constraints (2) cannot be satisfied, e.g. having the constraints $x \leq 1$ and $x \geq 2$ would lead to an infeasible

solution and thus no optimal solution exists.

Another case of nonexistent optimal solution is if the problem is *unbounded*. In this case all the constraints can be satisfied but the value of the objective function (1) can be arbitrarily large and the objective value goes to infinity (negative infinity if the function is to be minimized). An example would be an objective function with at least one variable and no constraints. [12]

2.1.3 An LP Example

Let's say that John wants to buy some berries and as much as he can afford and carry. There are two kinds of berries, red and blue. The red ones weigh 1 kg per liter and cost 10 coins. The blue berries weigh 1.2 kg per liter but cost only 7 coins. John has 80 coins, he can carry up to 10.2 kg (very precise) and he wants at least one liter of the red berries. If we let *red* be the amount of red berries in liters and *blue* be the amount of blue berries, then this can be modeled as:

maximize

$$red + blue \text{ (we want to maximize the amount of berries)}$$

subject to

$$\begin{aligned} red + 1.2\ blue &\leq 10.2, \text{ (make sure John does not buy more than he can carry)} \\ 10\ red + 7\ blue &\leq 80, \text{ (John can not buy for more than the coins he has)} \\ red &\geq 1, \text{ (at least one liter of the red berries)} \end{aligned}$$

The unique optimal solution to this model is for John to buy 4.92 liters of the red berries and 4.4 liters of the blue berries.

In this example John doesn't mind paying a bit extra as the cost of the berries is not reduced by the model. Thus, out of two solutions, with the same total amount of berries, John might choose the more expensive one. This could be fixed by adding a variable representing the price of the berries with the constraint $red + blue - price = 0$ (then the variable *price* will be equal to $red + blue$) and minimizing that variable in the objective function. Then the objective function would look something like:

$$red + blue - c \cdot price$$

where $0<c<1$ is a constant that restricts the importance of reducing the cost, i.e. John would not rather save his money than to buy more berries.

However, in this simple example it would not affect the solution, but it is more likely to if the problem was an IP problem.

2.2 Integer Programming

Linear programming is, as mentioned, a very useful tool so solve several optimization problems. Unfortunately, not all problems can be described using only real variables, e.g. half of a car and a third of a factory is rarely of any use

as a solution in real life. In these cases the variables of the model should only be able to assume discrete values. If we use the example from the LP section (see section 2.1.3), then it would be an IP problem if John could only buy the berries in whole liters, i.e. *red* and *blue* can only assume integer values.

Problems where some or all of the variables have this condition are called integer programming (IP) problems. If only some of the variables are required to be discrete then the problem is called a mixed integer programming (MIP) problem, otherwise it is called a pure IP problem. Using the way LP problems can be written with the added definition of which variables are discrete (see (3) below), an IP problem can be formulated in the following way:

maximize or minimize

$$z = \sum_{j \in N} c_j x_j$$

subject to

$$\begin{aligned} \sum_{j \in N} a_{ij} x_j + S_i &= b_i, \quad i \in M \\ S_i &\geq 0, \quad i \in M \equiv 1, 2, \ldots, m \\ x_j &\geq 0, \quad j \in N \equiv 1, 2, \ldots, n \\ x_j \text{ an integer}, &\quad j \in I \subseteq N \end{aligned} \tag{3}$$

where S_i are the slack variables. If $I = N$ then the problem is a pure IP, otherwise a MIP.

Just as with LP problems there either exists an optimal solution for an IP problem, in which case it is unique or one of infinitely many, or it is either infeasible or unbounded. However, the condition of discrete variables changes the characteristics of the solution space. One would think that it would become more defined and easier to solve as the "search" does not have to consider all the infinite solutions as in the continuous case. But the discrete condition does in fact destroy the attributes of the solution space used to solve LP problems [11, p.3].

One then might think that one can relax the integer constraint, solve it using LP and then round the solution to the nearest discrete values would give optimal solution. This is, unfortunately, not the case. The rounding does not necessarily lead to an optimal solution, it might not even lead to a feasible solution [13]. The discrete condition does in fact make the IP problem NP-complete [14] for which no polynomial time algorithm has been found.

Even in the special case of IP where the variables can only assume values 0 or 1, binary integer programming (BIP), is NP-complete. It is in fact one of Karp's 21 NP-complete problems [15]. Binary variables are maybe the most commonly used integer variables. It is practical to express yes/no decisions with, i.e. "Should we build a factory in Argentina?" where 1 can stand for yes and 0 for no. [16, p.76].

2.3 Solving IP Problems

So, how does one find an optimal solution to an IP problem? The most common method is the *branch and bound* method [17]. It performs a linear relaxation (LR) of the problem. That is, it disregards some or all of the integer restrictions and solves the problem with LP. The method then finds a fractional variable, say $x = 3.7$, of the LP solution and creates two new sub-problems, *branching*, by adding a new constraint for each. One constrains the variable to be greater than or equal to its rounded up value ($x \geq 4$) and the other to be less than or equal to its rounded down value ($x \leq 3$). The process is then repeated by solving each of these sub-problems with LP.

This method creates a tree of sub-problems which is in the size of 2^n where n is the number of integer variables. Since this can be quite a large number one wants to reduce the number of sub-problems which are considered. This can be done by *pruning*. The pruning is based on some consequences of the linear relaxation being less constrained than that of the original IP problem:

- If the IP problem is a minimization problem then the optimal solution value of the LR is less than or equal to the optimal solution value of the IP problem.
- If the IP problem is a maximization problem then the optimal solution value of the LR is greater than or equal to the optimal solution value of the IP problem.
- If the LR is infeasible then so is the IP problem.
- If LR has an integer solution (all integer variables are integer-valued) then that solution is optimal for the IP problem as well.
- If the coefficients of the objective function are integer-valued, then for minimization problems, the optimal solution value of IP is greater than or equal to the rounded up optimal solution value of the LR. For maximization problems, the optimal solution value of IP is less than or equal to the rounded down optimal solution value of the LR.

The above points are taken and reformulated from R. Bosch and M. Trick, Search Methodologies: Integer Programming [16, p.72]

Thus, before branching a problem one checks whether it is a feasible solution, if it is not then none of its descending problems will be it either, and so no further branching is needed. If it is feasible then let its LP solution value be compared to the best integer solution value obtained so far, if it is worse then so will its descending problems and so no further branching is needed. If the LP solution is an integer solution then it is an optimal solution for that subtree and no further branching is needed. The acquired integer solution is then compared to the best integer solution found so far and stored if it is better. This process is continued until no more subproblems exist.

A solution obtained by this method is thus an optimal one. However, if an optimal solution is not required then the algorithm can instead stop when the gap between the highest LR value (lowest if minimizing) and the best integer solution is sufficiently small. This is useful when the problem takes a long time

to solve.

The same problem can be modeled in several different ways and how the problem is formulated will affect the time it takes to solve it. One wants to find a formulation whose linear relaxation does not differ too much from the integer problem. If the linear relaxation gives an integer solution then there will not be any need for branching [16, p.79]. How to best formulate the problem is dependent on how the algorithm of the solver is constructed. But since the source code of most commercial solvers is unavailable, one has to rely on the advice of the solver makers. Or try different formulations oneself. In the end of this chapter 6 reformulations will be discussed.

However, one of the reformulations relies on special ordered sets, which is covered first.

2.4 Special Ordered Sets

Special ordered sets (SOS) where first introduced by E. M. L. Beale and J. A. Tomlin [18]. They are additional ways of specifying integrality conditions and are supported by many solvers. SOS can be flagged to the solver and are used in the branch and bound method for branching on sets of variables rather than individual variables. This helps the branch and bound method and will generally reduce the time taken to find a solution [19].

In the following sections SOS of type 1 and 2 will be covered. There are SOS of higher types but they will not be described here.

2.4.1 SOS Type 1

A SOS of type 1 is a set of variables at most one of which can take a non-zero value. It is mostly used with binary variables, i.e. where there is a choice between a set of options [20].

An example of this is if John (from section 2.1.3) was offered to buy one bag; a big or a small, that would enable him to carry a bit more berries. Lets say that the small bag can carry 3 more kg and costs 2 coins and that the big bag can carry 5 more kg and costs 4 coins. This can be modeled by modifying the constraints:

maximize

$$red + blue$$

subject to

$$\begin{aligned}
red + 1.2\ blue - bag_{carry} &\leq 10.2, \text{ (adding the carrying capacity of the bag)} \\
10\ red + 7\ blue + bag_{cost} &\leq 80, \text{ (adding the cost of the bag)} \\
red &\geq 1, \\
bag_N + bag_S + bag_B &= 1, \text{ (flagging the SOS type 1)} \\
2\ bag_S + 4\ bag_B - bag_{cost} &= 0, \text{ (setting the cost of the bag)} \\
3\ bag_S + 5\ bag_B - bag_{carry} &= 0, \text{ (setting the carrying capacity of the bag)} \\
bag_N, bag_S, bag_B &\in \{0, 1\},
\end{aligned}$$

where bag_N is the choice of not buying a bag, bag_S is the choice of buying the small bag and bag_B is the choice of buying the big bag.

2.4.2 SOS Type 2

A SOS of type 2 is a set of variables where at most two of them can take non-zero values. If two variables are non-zero then they have to be adjacent. Adjacency of two variables in a SOS type 2 is defined as no other variable having a weight value (which is declared for each variable together with the SOS) between theirs. SOS type 2 are generally used for modeling problems containing piecewise approximations of functions of a single variable [20].

For an example of this, we will once again return to John (see section 2.1.3). This time the price is reduced by 10% for the red berries if he buys 2 kg or more. This will make the cost of the red berries a function $P(n)$, where n is the amount of red berries, defined as:

$$P(n) = \begin{cases} 10n & \text{if } n < 2 \\ 9n & \text{if } n \geq 2 \end{cases}$$

By using SOS of type 2 this function can be modeled by modifying the constraints in the following way (assuming a max amount of red berries of 150 liters):

maximize

$$red + blue$$

subject to

$$\begin{aligned}
red + 1.2\ blue &\leq 10.2, \\
price_{red} + 7\ blue &\leq 80, \text{ (adding the cost of the red berries)} \\
red &\geq 1, \\
p_1 + 2p_2 + 3p_3 + 4p_4, &\text{ (flagging the SOS type 2 - the coefficients are the weights)} \\
0p_1 + 2p_2 + 2p_3 + 150p_4 - red &= 0, \text{(set the amount of red berries)} \\
0p_1 + 20p_2 + 18p_3 + 1350p_4 - price_{red} &= 0, \text{(set the price of red berries)} \\
p_1 + p_2 - line_1 &= 0, \text{(define line 1)} \\
p_3 + p_4 - line_2 &= 0, \text{(define line 2)} \\
line_1 + line_2 &= 1, \text{(the amount of red berries can only be on one of the lines)} \\
p_1, p_2, p_3, p_4 &\in [0, 1], \\
line_1, line_2 &\in \{0, 1\},
\end{aligned}$$

where a linear combination of two of p_1, p_2, p_3 and p_4 represents the quantity and price of the red berries. p_1 represents the point where 0 red berries are bought and thus cost 0 and p_2 represents the point where 2 kg is bought which would cost 20 coins if no reduction in price is applied. $line_1$ represents the line between these two points and any point on it represents an amount and its cost without the price reduction. $line_1$ is 1 if the quantity is between 0 and 2, otherwise 0.

$line_2$ represents the amount of red berries and their cost with the price reduction. It goes through point p_3, 2 kg and 18 coins (20*0.1=18), and point p_4, 150 kg and 1350 coins (1500*0.1=1350). $line_2$ is 1 if the quantity is between 2 and 150, otherwise 0. Exactly one of $line_1$ and $line_2$ must be 1.

There is nothing to represent a linear combination of p_2 and p_3 since they represent the same amount of red berries (but different prices).

Although the reduction in price is triggered when John buys more than 2 kg, when $line_2 = 1$, it is possible for it to not be triggered when he buys exactly 2 kg. Since point p_2 represents the same amount of red berries as p_3, then p_2 can be equal to 1 instead of p_3. Usually this is solved by the objective function minimizing the price of the berries (as explained in section 2.1.3). However, the objective function of this example does not have such a variable.

The constraint $line_1 + line_2 = 1$ is not really necessary in solving the problem. It emphasizes that the amount of the red berries is on exactly one of the two lines: the one representing no reduction in price or the one which represents the cost reduction of the red berries. Hopefully it helps in the understanding of this example.

2.5 Reformulations

The following section presents 6 ways of formulating an IP problem together with a way to reformulate them. The four first reformulations come from the company's way of modeling negative and positive variables. The last two reformulations (separating the constraints and triggered penalty reduction) were told by Mats Carlsson to Karl Sundequist[2] who in turn told them to me.

2.5.1 Change the Domain

The domain refers to the range of values of the variables. This reformulation changes the variables that are in the domain $[-1, 0]$ to the domain $[0, 1]$, i.e. all variables α:

$$\alpha \in [-1, 0]$$

has its domain changed:

$$\alpha \in [0, 1]$$

It is of course required to change the sign of α in the objective function and the constraints.

2.5.2 Change the Objective Function

The previous reformulation changed the sign of all the α variables. This reformulation is applied to the objective function to remove the acquired negative signs of these variables. If the problem is to maximize the objective function then it is changed to minimizing it and the reverse if the problem is to minimize the objective function. For example:

$$\text{maximize: } -\alpha_1 - \alpha_2$$

can be reformulated to:

$$\text{minimize: } \alpha_1 + \alpha_2$$

[2]Carlsson is a senior researcher at SICS, Uppsala, Sweden and Sundequist is a PhD student at the department of Information technology at Uppsala University, Sweden. They are both in an IP research project together with the company (see section **??**).

2.5.3 Change the Constraints

This reformulation handles the effects to the constraints of changing the sign. Similarly to the 'Changing the objective function'-reformulation, the constraints are changed so that the α variables will have a positive coefficient. If the constraint is an $\geq$ then it is changed into a $\leq$ and vice versa. For example:

$$-\alpha_1 - \alpha_2 - \alpha_3 \geq -\beta$$

can be reformulated to:

$$\alpha_1 + \alpha_2 + \alpha_3 \leq \beta$$

2.5.4 Allocation Representation

This reformulation changes what the *alloc* variable represents, from having the value of how much that is not allocated to how much that is allocated. For example:

$$\begin{aligned} \text{minimize: } & c \cdot alloc \\ \text{subject to: } & \alpha_1 + \alpha_2 + alloc \geq 1 \end{aligned}$$

can be reformulated to:

$$\begin{aligned} \text{minimize: } & c(1 - alloc) \\ \text{subject to: } & \alpha_1 + \alpha_2 - alloc = 0 \end{aligned}$$

2.5.5 Separating Constraints

If possible, this reformulation separates a constraint into several.

$$\alpha_1 + \alpha_2 + \alpha_3 \leq 3\beta$$

can be reformulated to:

$$\begin{aligned} \alpha_1 &\leq \beta \\ \alpha_2 &\leq \beta \\ \alpha_3 &\leq \beta \end{aligned}$$

where $\beta \in \{0, 1\}$.

2.5.6 Triggered Penalty Reduction

Reductions can be triggered if the size, a, of an allocated-penalty variable exceeds a certain limit, l. If triggered then the penalty which that variable applies to the objective value will be reduced. There are two kinds of reductions; lump reduction (LR) or percentage reduction (PR).

The LR gives a constant reduction of x to the objective value when $a \geq l$. The PR reduces the objective value depending on the size of a, i.e. let p be the penalty per unit of the variable ($a \cdot p =$ the total penalty) then if $a \geq l$ the reduction of the objective value is $d \cdot a \cdot p$, for some constant d, which is usually $d \in (0, 1)$. $d \leq 0$ would mean that the penalty is zero or that the allocated-penalty variable increases the objective value and $d \geq 1$ would mean that there is no reduction or that the penalty of a will grow as its size increases.

To represent a triggering of the reduction one can use a binary variable β which is 1 if the size of the allocated-penalty variable exceeds the limit l, otherwise it is 0. Let α_i be an allocated-penalty variable and c_i be the penalty for that variable, then an example of the PR can be represented as:

minimize

$$c_1\alpha_1 + c_2\alpha_2 + c_3\alpha_3 - d(c_1f_1 + c_2f_2 + c_3f_3)$$

subject to

$$\begin{aligned}
&\alpha_1 + \alpha_2 + \alpha_3 \geq l \cdot \beta, \\
&f_i \leq \beta, \\
&f_i \leq \alpha_i, \\
&\beta \in \{0, 1\}
\end{aligned}$$

for $i = \{1, 2, 3\}$. Where c_i is the penalty for variable α_i.

This can be reformulated to the way that the reduction is formulated in the example in the SOS type 2 section (see section 2.4.2). It can also be generalized to three dimensions by using a linear combination of four variables (instead of two) to represent all three dimensions, if needed. This is a little too complicated to be explained here.

3 The Project

In this section the actual project is described. It starts off with a brief about Xpress solver, the integer programming solver used which is followed by a short background of the project and a description of its progress and results. The company which provided the means and material for this project, will be referred to as 'C'.

3.1 Xpress

The Xpress solver is a product of the FICO© company which provides software for improving business decisions. The Xpress software itself is used for the development and deployment of optimization applications. The solver uses a multi-threaded optimization algorithm which can solve LP, MIP and several other kinds of problems. It works on most common computer platforms and provides several software interfaces in, among others, C, C++, Java as well as a standalone command-line interface 'optimizer'. Whenever 'the solver' is mentioned it is the Xpress software that is meant.

3.2 Project Background

C was presented with an IP problem, obtained from one of their customers, which had proven to be a bit harder to solve than most others. The C model did not find an optimal solution even after working on the problem for 24 hours. It was thus used for finding different pre-processing techniques to reduce the solving time.

The problem had been sent as a LP file to Karl Sundequist at the department of Information technology at Uppsala University, Sweden. He in turn tried different formulations of the problem and found 6 reformulations which provided the best solving time when using 'optimizer'. The reformulations are the alternative formulations found in the background (see section 2.5). With the 6 reformulations the solver found an optimal solution in less than a minute. Since the reformulations had not been solved using the C model it could not be ruled out that it was the solver's Java interface that slowed things down or that it was the model itself.

3.3 Project Description

In order to find out whether the model or the Java interface hampered in finding the solution, the reformulated problem had to be solved through the model. But the reformulated problem was described as a LP file which the model could not read. Converting the LP file manually into another format or hard coding it into the model was unreasonable as the file contained over 40 000 lines. Instead a parser was created (see section 4) which enabled the model to handle the reformulated problem in its original format.

4 Parser

This section describes the parser which was built and used for the project. It begins with a brief of a lp-file, its structure and some definitions, and then continues with how the parser handles the three major sections of it. The sections will be presented in the order that the parser handles them which is followed by an example of a lp-file.

4.1 The Lp-file

The lp-file format is a way to represent a problem to be solved by an IP solver. I only had access to a few examples of lp-files and of course the lp-files of the problem which I was to investigate. This meant that the parser was created with theses files in mind and that it can not handle all the instructions that a lp-file can have. Some web pages [21][22][23] were also used for cross-referencing.

The lp-files for the Xpress solver differ somewhat from the lp-files created by the model of C. The differences are not great but require the parser to be somewhat versatile. Luckily, no conflicting differences were found, i.e. no instructions in the Xpress solvers lp-files meant something else in the lp-files of the model. The parser also handles some instructions of the lp format for the CPLEX solver.

In most of the examples in this text the number, variable and instruction are often separated by a whitespace character. This is not required by the parser except where it is explicitly mentioned otherwise.

The lp-file consist of three sections; declaring the objective function, the constraints and the variables. It ends with the keyword `End` (see section 4.1.3), everything before the declaring of the objective function and after keyword `End` is ignored. The order in which these sections are encountered in a lp-file differs from the order that the parser handles them. When creating the model used by C one needs to state whether it is to maximize or minimize the objective function and that information is thus collected first. But before adding the objective function or any constraints the model requires that all the variables be known and declared. So the section of declaring variables is handled thereafter. This requires the lp-file to be read twice.

One could try to store the objective function in a data structure and the constraints in another. But since the lp-files can be quite large, e.g. one file had 191515 lines before reaching the variables-section, this would require a great amount of data storage. Data which already can be found in the lp-file, by reading it again.
Some definitions are needed for some of the parts of the lp-file, which are provided below.

4.1.1 Variables

There are some restrictions to the variable names so as not to confuse them with instructions to the parser. They can not start with a number since the coefficient might not be separated from the variable by a whitespace and they can therefore not being with a '`.`' nor an '`E`' followed by a number. Variable names can not contain the characters. '`=`', '`<`', '`>`', '`+`' or '`-`'. Nor can they begin

with `inf` or be any of the keywords used throughout the lp-file (see section 4.1.3).

4.1.2 Numbers

Numbers exist in the lp-file either as a coefficient to a variable (positioned before it) in the declaration of the objective function and the constraints, or as a bound when declaring the variables. A number in the lp-file is read and stored as a double. The parser supports scientific notation (E), for example `2.176E7` is interpreted as 21760000. A number beginning with a '`.`' is read as beginning with '`0.`', for example `.1234` is interpreted as 0.1234. Since the numbers are stored as double they may not exceed the limits in size of a double in Java.

4.1.3 Keywords

Throughout the lp-file there exist keywords which instruct the parser how to interpret the information. They can for example indicate the beginning of a section or describe how to handle the bounds of a variable. No regard is taken to upper and lower casing of the characters in a keyword, except where explicitly mentioned otherwise. However, if a keyword contains a white space then it should have exactly one in that position when used in the lp-file.
The keywords are:

```
Maximize     Maximise     Maximum      Max
Minimize     Minimise     Minimum      Min
Subject to   Subj to      Such that    s.t.        s. t.       st
Bounds
General      Gen          Gens
Integer      Int          Ints
Binaries     binary       bin          bins
End
SOS
SOS1
SOS2
S1
S2
+infinity    +inf         infinity     inf
-infinity    -inf
free
```

4.1.4 Comments

A comment line in the lp-file begins with either '\' or '\\' and the parser treats the whole line as a comment. They can occur at any line in the file. When preceding a bound or a constraint declaration the commented lines are collected and added to the model together with the variable and constraint respectively. If there are several lines of comments, separated only by empty lines, then they are all concatenated and stored as a long string. If the bounds of a variable are declared on separate lines (see section 4.2.1), each preceded by a comment, then those two comments are concatenated and stored with the variable. Comments preceding a variable declaration in another section than the Bounds section

(see section 4.2.3) are ignored and so are comments within a constraint, i.e. a commented line in between two lines of a constraint declaration.

4.1.5 Labels

A label can precede the objective function or the declaration of a constraint. An example could be:

```
hello: x0 + x1 <= 0.0
```

where 'hello:' is the label. These are ignore by the parser, i.e. anything preceding and including ':' on the line is disregarded.

4.2 Declaring the Variables

Adding the variables of the lp-file to the C model is done with three methods depending on whether the variable is a real, integer or a binary. If it is a real or an integer then an upper and lower bound is to be provided as arguments. A comment can be provided for all three methods or set to null if none is wanted. The methods return an instance of the model's class Variable which is to be used when adding that variable to the objective function or a constraint. These instances of Variable are stored together with the variable name it has in the lp-file, as a reference, so it can be fetched and used later.

4.2.1 Declaring Bounds

The declaring of the bounds starts with the Bounds section which begins with the keyword `Bounds` and the parser assumes that the lp-file has one. However, the Bounds section may be empty. It is in this section that the bounds of the variables are declared. The parser is unaware of which type the variable is when its bounds are declared in the file and thus the bounds are stored together with its variable name and comment (if one exists). These are not added to the model until a later time.

It is assumed that for each declaration the variable does not have a coefficient (or rather a non-written coefficient of 1.0) and that the upper and lower bounds are numbers, i.e. they can not be variables.

The declaring of the variables can be done in several ways, either by declaring both upper and lower bound on the same line, e.g.

```
.31 <= z1 <= 2E2
```

which sets the lower bound of the variable z1 to 0.31 and the upper to 200.0, or by writing the declaration of the upper and lower bound on different lines (doesn't need to be consecutive or in order). Thus:

```
z1 <= 200
31E-2 <= z1
```

is equivalent with above.

However, the declaration of a lower bound, upper bound or both can only be on one line of the lp-file, i.e. a newline character should not appear before

the declaration of a bound is done.

The relation symbol `>=` can also be used to declare bounds. The previous example can be written as:

```
200.0 >= z1 >= 0.31
```

These lower and upper bound declarations can also be written on separate lines.

Declaring using `=` is also allowed and sets both bounds to the same number. Thus for:

```
z0 = 42
```

both the upper and the lower bound of the variable z0 is set to 42.0.

However, switching the sides of the equal sign is not allowed, e.g. declaring a variable with `42 = x1` would raise an error.

4.2.2 Non-existent Bounds

Although a bound can not be a variable, it can be declared with the keyword `inf` or `infinity` with an optional `+` or `-` sign preceding it. This represents a non-existent bound. For example

```
0 <= x0 <= infinity
-inf <= x1 <= 4E+3
```

means that the variable x0 is bounded below by zero but has no upper bound, and that the variable x1 has no lower bound but is bounded from above by 4000.

If the variable does not have either bound then the keyword 'free' can be used. For example:

```
y0 free
```

is the same as

```
-infinity <= y0 <= infinity
```

A restriction to using non-existent bounds is that the lower bound can not be positive infinity and the upper bound can not be negative infinity. Thus a variable can not be set equal to positive/negative infinity. When a variable with a non-existent bound is added to the model then two special number are used as arguments to represent positive or negative infinity.

4.2.3 Default Bounds

The defining of variables has three subsections; General, Integer and Binaries. They follow after the Bounds section but neither of them is necessary for the parser and can come in any order. These sections contain just variable names, one per row, and no bounds. A variable in the Bounds section may also be in one of General, Integer or Binaries but not in two or more of them. If a variable's upper or lower bound is not defined in the Bounds section then the

default value is used. The default value is dependent on which of the sections the variable is defined in.

The General section begins with the keyword `General`, `Gen` or `Gens`. All variables defined here have default bounds [0, infinity] and are added as integers to the model.

The Integer section begins with keyword `Integer`, `Int` or `Ints`. All variables defined here have default values [0, 1] and are added to the model as integers.

The Binaries section begins with keyword `Binaries`, `Binary`, `Bin` or `Bins`. The variables defined here are added to the model as binaries and will thus get the lower bound of 0 and the upper bound of 1.

If a variable is only defined in the Bounds section then the default bounds are [0, infinity] and it will be added to the model as a real. However, a variable can not be defined with just its variable name in the Bounds section; it has to have at least one declared bound.

The lp-file can contain sections for semi-continuous, semi-integer and partial integer but the model does not support the use of these and they are not allowed by the parser.

The section of declaration of variables ends with keyword `End` which must exist at the end of the lp-file.

4.3 Declaring the Objective Function

When the variables have been collected and stored within the model then so can the objective function be. It is assumed that an objective function exists and that it is not empty, i.e. it contains variables, and that the variables in it are declared in the lp-file. The objective function can be preceded by the keyword `Maximize`, `Maximise`, `Maximum` or `Max` indicating that it is to be maximized. Or it can be preceded by `Minimize`, `Minimise`, `Minimum` or `Min` indicating that the objective function is to be minimized. If such a keyword precedes the objective function then it should be the first line with instruction of the file, i.e. the first non-empty and non-commented line. If no such keyword exists then the objective function is assumed to be maximized.

The objective function in the lp-file is made of terms which consist of a variable with an optional number coefficient preceding it. These terms are separated by the operators addition, `+`, and subtraction, `-`. When adding the objective function to the model an instance of the models Linear Combination class is created. One then calls a method of that class to add a term of the objective function. The method requires a number as a coefficient and an instance of the model class Variable (see section 4.2).

If a variable does not have a coefficient preceding it in the file then it is assumed to be 1.0 if the previous operator was addition or if there was no previous operator (for the first term). The coefficient is assumed to be -1.0 if the previous operator was subtraction. No other operators are permitted, e.g. multiplication, division, exponentiation, etc. Neither is parenthesis.

If a term in the objective function would consist of only a number, lacking variable, then the parser will add a variable to the model with its lower and upper bound set to that number. The parser will not take into account if the same variable occurs more than once in the objective function and it will thus be added several times to the model.

Since the objective function can be quite large it is often divided on several lines. The parser makes no assumptions of where the line break will occur as long as it does not divide a variable name or a coefficient into two. The line break can thus come before or after an operator or even between a variable and its coefficient. Any comment lines in between the lines of the objective function are ignored.

The section for declaring of the objective function ends with the keyword indicating that the section for the declaring of the constraints begins, i.e. `Subject to`, `Subj to`, `Such that`, `s.t.`, `s. t.` or `st`. When the parser reaches this keyword it adds the instance of Linear Combination to the model.

4.4 Declaring the Constraints

For the model it does not make any difference whether the objective function is added before, after or between the constraints. Since the constraints follow the objective function, in the lp-file, it is more convenient for the parser to handle them in that order. The declaring of the constraints is preceded by the keyword `Subject to`, `Subj to`, `Such that`, `s.t.`, `s. t.` or `st`. It is assumed that a constraint section exists, although it can be empty.

Just as with the objective function the constraints in the lp-file are made of terms consisting of a variable with an optional number coefficient preceding it. The terms are separated by the operator addition and subtraction. And when adding a constraint to the model an instance of the models Linear Combination class is first created to which the terms are added. The interpreting, collecting and adding the terms are done in the same way as with the objective function.

However, the constraints differ from the objective function in that they end with a relation symbol followed by a number. The right side of the relation symbol can not be a term but must be a number which can not be positive or negative infinity. A line break occurring before or after the relation symbol is handled by the parser. The relation symbol is one of `<=`, `>=`, or `=`. A different method for adding the constraint to the model is used for each of the relation symbols. All three methods require an instance of the Linear Combination class, the number on the right side of the relation symbol and an optional comment (set to null if it is not wanted).

If the constraint is a special ordered set (see section 2.4) then this is defined in the lp-file with the relation symbol `=` followed by the keyword `S1` or `S2` depending on whether it is a special ordered set of type 1 or 2 respectively. The model does not support special ordered sets of higher order than type 2.

There are other ways that the special ordered sets can be declared in the lp-file. They can, for example, have their own section which is preceded by keyword `SOS` and should follow the declaration of variables. The constraints in this section begins with `S1` or `S2` followed by a double colon `::` to indicate which type of special ordered set it is.

The lp-file can also contain different sections for the type 1 and 2 of the special ordered sets. These sections are then preceded by keywords `SOS1` and `SOS2`

respectively and follow somewhere in or after the declaration of the constraints in the lp-file. The constraints in these sections are made of terms separated by commas ’,’. However, the parser has not been implemented to support these types of declarations of special ordered sets.

The section for declaring the constraints ends with the keyword `Bounds`, indicating that the section of declaring the variables begins. Since this section is handled first by the parser it means that the whole lp-file has been read and that the parser is done.

4.5 Example

Using the example from the SOS section (see section 2.4.2) then a lp-file would look something like this:

```
max
red + blue

subject to
\\ Weight restriction - John can not carry more than 10 kg
1 red + 1.2 blue <= 10
\\ Must buy at least 1 kg red berries
red  >= 1
\\ Price - John can not pay more than 80
priceRed + 7 blue <= 80

p1 + 2 p2 + 3 p3 + 4 p4 = S2
0 p1 + 2 p2 + 2 p3 + 150 p4 - red = 0
0 p1 + 20 p2 + 18 p3 + 1350 p4 - priceRed = 0

p1 + p2 - line1 = 0
p3 + p4 - line2 = 0
line1 + 2 line2 = S1

Bound
\\ Red berries
0 <= red <= 150
\\ Blue berries
blue free
\\ Price of the red berries
redPrice >= 0

Integer
blue
p1
p2
p3
p4

Binaries
line1
line2

End
```

5 Results and Conclusion

The difference in time when finding an optimal solution with optimizer and solving it through the model was insignificant. Optimizer took 24 seconds and the model took 49 seconds (including the time it took to parse the LP file).

It was thus the 6 reformulations which brought on the great reduction in time. Furthermore, according to Karl Sundequist, two of them had greater effect than the others. The two reformulations were a) separating the constraints (see section 2.5.5) and b) triggered penalty reduction (see section 2.5.6). The first of these two was selected for more testing and the original problem was altered by using only this reformulation. This new constraint-separated-problem took only 160 seconds to solve using the model. Thus, it was assumed that this one reformulation was responsible for most of the time reduction and that it should be investigated further.

It should be noted that separating the constraints was shown to decrease the performance of the solver for some other problems at C.

The customer that provided the first problem had expanded it somewhat by changing a few of the constraints as well as adding some new ones. Thus this expanded problem shared a lot of its characteristics with the original one. But when altering it with the constraint-separating reformulation, it still timed out after 2000 seconds (about half an hour), i.e. no optimal solution was found within the time frame.

The same drastic time reduction as for the original problem was not present. However, gap between the best linear relaxation value and the best integer solution at the timeout was approximately 95331 when not using the reformulation and 42389 when using it. It is more than half the gap and, although no optimal solution was found, can be seen as a great improvement.

The changes between the original file and the expanded file were investigated; 3 constraints had been altered and 9 new ones had been added. Unfortunately there was not enough time to thoroughly analyse the changes to find out which of the constraints and what combination of them it was that hampered the reduction in time.

Solving complex IP problems is important for many companies in order to help them with their business decisions. For some companies it is the answer that is important, finding an optimal solution, and for others it is how fast they can get a reasonable estimate.

In this case the reformulation has achieved both goals, making the solver find an optimal solution within minutes. Knowing that it works for some problems means that a user has an extra way of formulating the problem in case the first did not give a good performance. But, the reformulation does not work for all sets of problems and can actually worsen the performance of the solver. It is thus not a clear choice as a standard formulation. However, with the computer power of today one can solve the problem both with and without the reformulation, at the same time, and then derive the solution from whichever finishes first.

Another fortunate outcome of this project was that the reformulation did not work for the expanded problem, which does not differ much from the original problem, for which the reformulation did work. This means that the changed

and the added constraints can be analysed to find out which of them affect the time reduction of the reformulation and hopefully identify attributes of the problem so that they can be found in other problems. This could be used to pre-set the model to automatically find better formulations of problems, or even to be used by a solver to, for example, relax those constraints in the initial stage of solving.

References

[1] S. I. Gass, 2003, *Linear Programming: Methods and Applications*, 5th ed, p.3. New York: Dover Publications, Inc.

[2] J. C. Nash, 2000. *The (Dantzig) simplex method for linear programming*, Computing in Science and Engineering, Vol.2, No.1, pp.29-31.

[3] V. Klee and G. J. Minty, 1972. How good is the simplex algorithm?, O. Shisha (editor), *Inequalities III*, New York: Academic Press, pp.159-175.

[4] J. W. Chinneck, 2001. *Practical Optimization: a Gentle Introduction*, [pdf], Ch.7, p.1. Ottawa: Carleton University. Available at: <www.sce.carleton.ca/faculty/chinneck/po.html> [Accessed 15 August 2012].

[5] M. C. Delfour, 2012, *Introduction to Optimization and Semidifferential Calculus*, Preface, p.xiii. New York: Society for Industrial & Applied Mathematics.

[6] A. Charnes, W. W. Cooper and B. Mellon, 1952. *Blending Aviation Gasolines - A study in Programming Interdependent Activities in an Integrated Oil Company*, Econometrica, Vol.20, No.2, pp.135-159.

[7] G. B. Dantzig, 1998, *Linear Programming and Extensions*, p.10. West Sussex: Princeton University Press.

[8] R. E. Kalaba and M. L. Juncosa, 1956. *Optimal Design and Utilization of Communication Networks*, Management Science, Vol. 3, No.1, pp. 33-44.

[9] V. Chvátal, 1983. *Linear Programming*, p.8, New York: W. H. Freeman and Company.

[10] T. S. Ferguson, unknown year. *Linear Programming A Concise Introduction* [pdf], p.3. Available at: <www.math.ucla.edu/∼tom/LP.pdf> [Accessed 14 August 2012].

[11] H. A. Taha, 1975. *Integer Programming: Theory, Applications, and Computations*. London: Academic Press, Inc.

[12] J. Lundgren, M. Rönnqvist and P. Värbrand, 2001. *Linjär och ickelinjär optimering*, p.22. Lund: Studentlitteratur.

[13] G. E. Thompson, 1971. *Linear Programming: An Elementary Introduction*, p.210. New York: The Macmillan Company.

[14] J. von zur Gathen and M. Sieveking, 1978. *A Bound on Solutions of Linear Integer Equalities and Inequalities*, Proceedings of the American Mathematical Society, Vol.72, No.1, pp.155-158.

[15] R. M. Karp, 1972. Reducability Among Combinatorial Problems. R. E. Miller and J. W. Thatcher (editors), *Complexity of Computer Computations*, New York: Plenum, pp.85-103.

[16] R. Bosch and M. Trick, 2005. Integer Programming E. K. Burke and G. Kendall (editors), *Search Methodologies*, New York: Springer.

[17] W. Zhang, 2008. *Parallel multi-Objective Branch and Bound*, Kongens Lyngby: Technical University of Denmark, p.9.

[18] E. M. L. Beale and J. A. Tomlin, 1970. Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables, J. Lawrence (editor), *Proceedings of the fifth international conference on operational research*, London: Tavistock Publications, pp.447-454.

[19] J. A. Tomlin, 1988. *Special ordered sets and an application to gas supply operations planning*, Mathematical Programming, Vol.42, No.1-3, pp.69-84.

[20] C. Gueret, C. Prins and M. Sevaus, 2000. *Applications of optimization with Xpress-MP*, Paris: Editions Eyrolles, p.40.

[21] Sourceforge. *LP file format* [online] Available at: <http://lpsolve.sourceforge.net/5.5/lp-format.htm> [Accessed 1 July 2012].

[22] Rensselaer: Department of Mathematical Sciences. *LP File Format* [online] Available at: <http://eaton.math.rpi.edu/cplex90html/reffileformatscplex/reffileformatscplex5.html> [Accessed 1 July 2012].

[23] Mosek, 2012. *The MOSEK Python API manual, Version 6.0, The LP file format* [online] Available at: <http://docs.mosek.com/6.0/pyapi/node022.html> [Accessed 1 July 2012].